

Giotto

Christian Buckl

TU München, 85748 Garching b. München, Germany,
buckl@in.tum.de,
WWW home page: <http://www6.in.tum.de>

1 Giotto

Giotto [1] is a programming methodology for embedded control systems and provides a time-triggered programming language, a compiler and a run-time system. The major goal of Giotto is to raise the level of abstraction, but to allow an unambiguous specification of the software architecture including a precise definition of timing. The specific implementation of the system on a specific platform is abstracted by the use of a run-time system. The compilation process guarantees a preservation of functionality and timing.

The main concepts of the time-triggered programming language are an actor-oriented design and the use of logical execution times as model of computation. The application is defined as a set of actors, the so-called *tasks*, that encapsulate the application functionality. Tasks are periodically executed functions that can for instance be implemented in C. A Giotto program consists of a set of *modes* that in turn consist of a set of tasks. By switching between different modes, tasks can be added or removed. At each point in time, the program is exactly one mode is active.

The execution of tasks is based on the simple task model [2]. This model excludes internal synchronization points. The only mean for communication between tasks are *ports*. The communication between ports of different tasks, as well as with sensors and actuators is performed by *drivers* that transport the values between ports. In contrast to tasks that consume a non-negligible execution time, the drivers are assumed to be executed instantaneously and therefore satisfy the synchrony assumption of synchronous languages [3].

The execution of drivers is performed time-triggered following the concept of *logical execution times* [4]. Figure 1 points out the general concept. Instead of using the physical execution time, all task computations logically take a fixed amount of time [5]. The frequency of the task execution is defined within the mode. At the logical start of a task, the values of relevant ports are copied to the task's input ports. At the logical end of the task execution, the results are published in the output ports. Using this concept, the physical execution of the task on the CPU becomes transparent to the application developer and can be managed by the run-time system.

The Giotto program can be compiled both to platforms using preemptive priority-based scheduling, as well as to time-triggered platforms. The mapping performed by the compiler can be simplified by annotating the program with

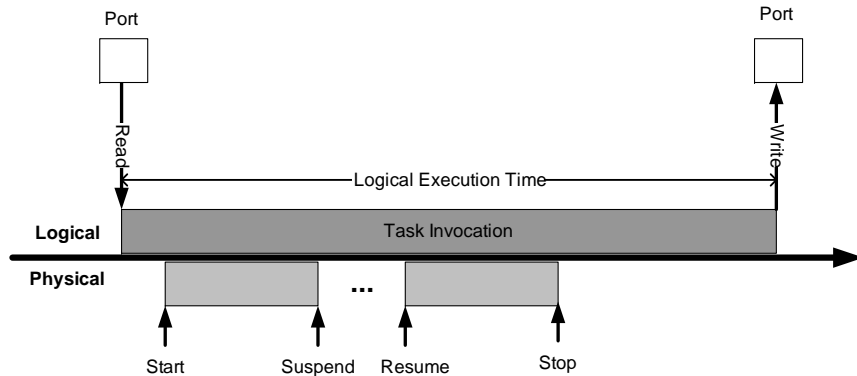


Fig. 1. Concept of Logical Execution Times

compiler directives, so called *platform constraints*. However, these constraints do not influence the functionality or timing of the application.

The execution of the generated code is performed by virtual machines to achieve the goal of platform independence. The result of the compilation process of the Giotto program is embedded code (E-Code). This E-Code is executed on a corresponding virtual machine, the E-Machine [6]. For scheduling, the E-Machine may pass the tasks to the scheduler of the operating system. Alternatively, it may use another virtual machine, the S-Machine, for scheduling. The S-Machine interprets scheduling code (S-Code) [7]. This approach has the advantage that the platform independent application execution (E-Code) and physical execution (S-Code) are separated [8]. The approach was demonstrated by a control system for the Giotto [5]. Furthermore, a Giotto program can also be compiled on distributed platforms [9].

Several research projects build upon Giotto. The **Timed Definition Language (TDL)** [10, 11] adds a component model and full support for distribution. The component model allows the decomposition of the whole program into distinct subprograms. The different components, called modules, can have different active modes, thus resolving the restriction of Giotto of only one active mode at a time. However, a module is limited to one hardware node. Mode changes affecting several nodes have to be realized by the developer. To simplify the application design, the code for drivers is generated in TDL by the design tools. Sensor and actuator functions must still be implemented by the application developer.

To support distribution, TDL allows a mapping of modules to hardware nodes, denoted as electronic control units (ECU) and generates the code for the time-triggered communication protocol FlexRay [12].

The **Hierarchical Timing Language (HTL)** [13] is another successor of Giotto. Similar to TDL, it adds the concepts of modules to support different modes to be active at a time. However, these modules are again restricted on one node. Communicators are used to realize the communication between

the different modules. This concept can be interpreted as global ports, whereas Giotto uses local ports attached to a task. The communication between different computational nodes should be supported by the tool, but must be currently implemented by the developer.

The development tool **FTOS** [14] is based on different ideas introduced in Giotto and targets the area of fault-tolerant real-time systems. In contrast to the virtual machine approach of Giotto, the approach is based on the generation of a tailored run-time system. The run-time system implements a correct scheduling scheme, the communication within the distributed system, I/O operations, distributed mode changes as well as fault-tolerance mechanisms such as active replication or rollback recovery.

References

1. Henzinger, T., Horowitz, B., Kirsch, C.: Giotto: A time-triggered language for embedded programming. *Proceedings of the IEEE* **91** (January 2003) 84–99
2. Kopetz, H.: *Real-Time Systems, Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Dordrecht, Netherlands (1997)
3. Benveniste, A., Caspi, P., Edwards, S.A., Halbwegs, N., Guernic, P.L., de Simone, R.: The synchronous languages 12 years later. *Proceedings of the IEEE* **91**(1) (January 2003)
4. Henzinger, T.A., Kirsch, C.M., Sanvido, M.A., Pree, W.: From control models to real-time code using giotto. *IEEE Control Systems Magazine* **23**(1) (2003) 50–64
5. Kirsch, C.M.: Principles of real-time programming. In: *EMSOFT '02: Proceedings of the Second International Conference on Embedded Software*, London, UK, Springer-Verlag (2002) 61–75
6. Henzinger, T.A., Kirsch, C.M.: The embedded machine: Predictable, portable real-time code. *ACM Transactions on Programming Languages and Systems (TOPLAS)* **29**(6) (2007)
7. Henzinger, T.A., Kirsch, C.M., Matic, S.: Schedule-carrying code. In: *Proceedings of the Third International Conference on Embedded Software (EMSOFT)*, Lecture Notes in Computer Science, Springer-Verlag (2003)
8. Kirsch, C.M., Sanvido, M.A.A., Henzinger, T.A.: A programmable microkernel for real-time systems. In: *VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, New York, NY, USA, ACM (2005) 35–45
9. Henzinger, T.A., Kirsch, C.M., Matic, S.: Composable code generation for distributed giotto. In: *Proceedings of the International Conference on Languages, Compilers, and Tools for Embedded Systems*, ACM Press (2005) 21–30
10. Farcas, E., Farcas, C., Pree, W., Templ, J.: Transparent distribution of real-time components based on logical execution time. In Paek, Y., Gupta, R., eds.: *LCTES*, ACM (2005) 31–39
11. Farcas, C., Pree, W.: A deterministic infrastructure for real-time distributed systems. In: *OSPERT 2007 Workshop on Operating Systems Platforms for Embedded Real-Time applications*. (2007)
12. Farcas, E.: *Scheduling Multi-Mode Real-Time Distributed Components*. PhD thesis, Department of Computer Sciences, University of Salzburg (Jun 2006)

13. Ghosal, A., Henzinger, T.A., Iercan, D., Kirsch, C.M., Sangiovanni-Vincentelli, A.: A hierarchical coordination language for interacting real-time tasks. In: Proceedings of the 6th ACM International Conference on Embedded software, Seoul, Korea, ACM (Oct 2006) 132–141
14. Buckl, C.: FTOS: Model-based development of fault-tolerant real-time systems. Technical report, Technische Universität München (2008)